Published in Proceedings of VSMM 2012: 18th International Conference on Virtual Systems and Multimedia, pages 267-274; definitive version available from http://dx.doi.org/10.1109/VSMM.2012.6365934

A Modular Framework for Deformation and Fracture using GPU Shaders

Derek John Morris and Eike Falk Anderson and Christopher Peters

Interactive Worlds ARG
Coventry University
Coventry, UK

Advances in the graphical realism of modern video games have been achieved mainly through the development of the GPU (Graphics Processing Unit), providing a dedicated graphics co-processor and framebuffer. The most recent GPU's are extremely capable and so flexible that it is now possible to implement a wide range of algorithms on graphics hardware that were previously confined to the computer's CPU (Central Processing Unit). We present a modular framework for real-time simulation of deformation and fracture for use in computer games that, rather than employing a General Purpose GPU (GPGPU) Framework, implements aspects of the simulation within shader programs on recent GPU's. Keywords-Real-time Graphics, Programmable Shaders, Deformations.

I. I NTRODUCTION

Modern computer games aim to immerse players in virtual game worlds that employ 3D animated graphics to provide players with an illusion of realism. A major contributing factor to this end has been a steep rise in the quality of real-time computer graphics, fuelled by dramatic advances in computer graphics hardware combined with the evolution of algorithms and methods that more realistically model the real world.

One particular area that contributes significantly to the perception of a realistic and immersive virtual environment, independent of its application to games [1], [2] or visual effects [3], [4], is the behaviour of object materials when other objects collide with them or they are destroyed. Examples include the shattering of a glass window pane as it is riddled with gunfire, or the crumbling of a brick wall as it is smashed with a wrecking ball. Traditionally, work on these effects has been based on CPU implementations, due originally to the lack of dedicated graphics hardware and also the fact that the required physics based algorithms require memory scatter and gather techniques that are much more straight-forward to implement on a CPU due to the nature of the hardware design. However, as GPU technology has evolved over the years, traditionally fixed aspects of the hardware accelerated graphics pipeline have become programmable, allowing a much larger range of algorithms to be implemented. With the advent of technologies such as vertex and pixel shaders, multiple render targets (MRT's), and more recently geometry shaders and a stream out mechanism, it has become possible to treat the GPU more as a general purpose data stream processor [5], therefore enabling the migration of previously CPU based algorithms onto the GPU.

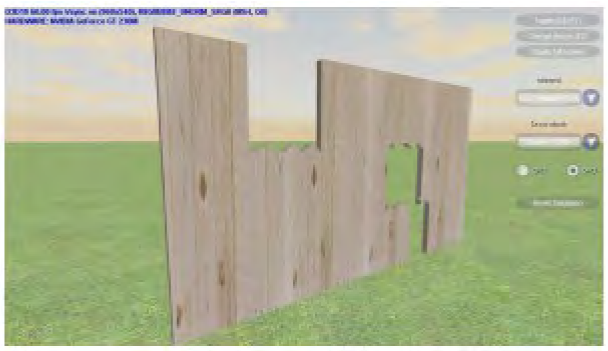

Fig. 1. A screenshot of our implementation. The image shows procedurally generated edges on a section of fractured wooden fence.

We present a platform (hardware vendor) independent, modular framework for real-time simulation of deformation and fracture in a video game setting (Figure 1). Our simulation makes use of a tetrahedral-based FEM incorporating the linear elasticity model and integrates motion using the explicit Euler method [6]. In its current form, our implementation disregards the effects of plasticity and the issues of warping under large rotational deformations. As the algorithms for deformation and fracture are modular and can be implemented on both the CPU and GPU, this opens the possibility to investigate effective load balancing of the overall system by distributing work-load across different processors. In order to achieve this, previously CPU-based algorithms are migrated to the GPU, utilising the latest hardware features, with options to switch between CPU or GPU processing for different sections of the simulation. Rather than using a GPGPU approach for implementing the simulation steps on the GPU, our system implements these using shader programs as found in the modern programmable graphics pipeline, applying a novel method of connectivity of finite elements used to simulate material fracture on a GPU stream processing architecture. This paper is organised as follows: in Section II, an overview is provided of the related work capabilities of modern GPU's that our system is based on. In Sections III and IV we describe the methods and techniques used for simulating deformation and fracture of materials, followed by a discussion of how these have been adapted to fit into our solution. Finally we present some simulation results in Section V and discuss possible directions for future work in Section VI.

II. BACKGROUND AND RELATED WORK

The initial drive to develop dedicated graphics acceleration hardware aimed to improve the performance of graphics rendering and was achieved by performing common tasks in hardware circuitry, starting with geometry processing [7] and building up to a complete rendering pipeline for texture-mapped, shaded scenes created from polygonal objects [8]. In the mid 1990's, sections of the graphics rendering pipeline began to be implemented in consumer graphics hardware and in 1999 NVIDIA released the first graphics card supporting hardware vertex processing (GeForce256) [9]. Recent graphics hardware has gained a number of new additions that enhance GPU capabilities. A programmable geometry shader unit has been added that allows more control over the processing of geometrical primitives. Stream out functionality allows data output from either the vertex shader or geometry shader stages to be re-routed to an external buffer. This allows processed data to feed back into the start of the pipeline, setting up a processing feedback loop. Finally, pixel data from the pixel shader stage can be output to a number of buffers simultaneously using MRT's, providing the ability to process and output separate sets of pixel data in a single rendering pass. A number of higher level shading languages were developed to encapsulate some of the underlying hardware complexity in the form of languages similar to the 'C' programming language including the languages Cg [10], HLSL [11] and GLSL [12]. With the advent of these flexible graphics hardware advancements, general-purpose computation on graphics hardware has become a popular domain of research and development. Real-time physics using GPGPU techniques have been implemented with fragment shaders by Joselli et al. [13], collision detection on the GPU has been achieved using GPGPU frameworks and programming languages [14], [15] and GPGPU solutions for volumetric deformations aimed at off-line processing for feature animation visual effects have recently been presented by Aldrich et al. [16]. A CPU-based solution for the handling of real-time deformation and fracture within a video game environment was presented by Parker and O'Brien [1], and released as commercial middleware containing libraries and tools called Digital Molecular Matter (DMM) [17]. In this system tools provide artists with control over the creation of destructible materials and allow the editing of simulation parameters, while the simulation itself has been implemented to run across multiple parallel CPUs to maximise performance (version 2 of the system uses GPGPU accelleration for some, but not all of its features). The system has been integrated into top selling video games including the game 'Star Wars: The Force Unleashed' [18].

Similarly, the recent “APEX Destruction” toolkit [19] provides a high-level authoring tool for destructible materials in which rigid bodies are implemented using GPGPU physics acceleration, with fracture information supplied by artists (as slicing information or in form of a ‘fracture map’).

III. M ATERIAL SIMULATION

Deformation and fracture of materials in computer graphics is greatly influenced by physical models originally developed for engineering. For real-time simulations, these physical models are then simplified as much as possible to maintain a favourable balance between performance and accuracy. In the area of real-time computer graphics, it is usually acceptable to reduce physical accuracy as long as the resulting visual representation remains plausible for observers [20]. The leading research on deformable models derives from elasticity theory in continuum mechanics which models the deformation of solid objects by calculating the internal stresses and strains generated from applied loads. The simplification of elasticity theory most commonly used in real-time computer graphics is linear elasticity theory which maintains a linear relationship between the stresses and strains and is therefore less computationally expensive than its nonlinear counterpart. In order to apply these physical models to a computer simulation they are numerically integrated across discrete timesteps to build up a convincing representation of motion over time [6].

A. Material Deformation

Terzopoulos et al. provide a good overview of the application of elasticity theory to model deformable objects in the field of computer graphics [21], describing the analysis of deformation as the calculation of the relative distances between all points within a solid volume. This is achieved using a Partial Differential Equation (PDE) that represents the entire material in a continuous fashion. The PDE is then discretised using the Finite Difference Method (FDM) that describes the material as a regular grid of nodes and evaluates the Ordinary Differential Equations (ODEs) resulting from this transformation between the grid nodes. The numerical integration of this solution governs the shape and movement of the material to be modelled. An improvement to the method of discretisation in order to allow irregularly shaped materials to be represented as made by O’Brien and Hodgins [22] and O’Brien et al. [23]. Here the Finite Element Method (FEM) is used and the material is split into tetrahedral shapes across its volume leading to the ODEs being evaluated across the tetrahedral faces. An algorithm for the FEM for modelling deformation in real-time computer graphics is described by Muller et al. [6]. A particular implementation issue with deformation simulations relates to an artefact that occurs with the linear FEM under large rotational deformations [9], [1]. The problem is known as stiffness warping and occurs because current methods rely on constant stiffness matrices that depend singularly on the rest configuration of the simulated material. The issue is addressed by explicitly extracting the rotational part of the deformation and thus calculating translations and rotations separately.

B. Material Fracture

When stresses exceed a certain threshold, materials in the real world either stretch out of shape before cracking and breaking apart or immediately crack and break apart based on the properties of the material. Existing research on fracture concerns itself with the modelling of this stretching effect using the model of plasticity and also the generation and propagation of the cracks, which in turn lead to the separation of the material into discrete entities. Fracture in the domain of real-time computer graphics is based on these physical models with the emphasis put on the visual representation of the effect rather than physical accuracy. O’Brien and Hodgins [22] introduce a method for analysing the stresses calculated in the deformation process, based upon the theory of linear elastic fracture mechanics, which determines where cracks should begin and how they should propagate throughout a simulated material. At each point in the material, a tensor is constructed describing the direction and size of the forces acting to break apart the material at that location. When the forces exceed a certain threshold, a fracture plane is computed perpendicular to the largest force direction and the material is separated along this plane. In this model the effects of plasticity before cracks appear are ignored and therefore it is used to simulate 'brittle' materials where cracks have a tendency to immediately appear after the stress threshold is reached. This fracture model has been refined to include the effects of plasticity for materials that require this [23], which is dubbed 'ductile' fracture. Here the strain element of the deformation model is decomposed into two elements: the strain due to plastic deformation and the strain due to elastic deformation. The behaviour of the plasticity element is described by a yield value and a description of plastic flow. As a result materials that exhibit 'ductile' properties follow elasticity principles up to the yield value, then stretch out of shape according to the plasticity properties and finally crack and break apart upon reaching a certain force threshold. Muller et al. describe the process of analysing the stresses from the stress tensors calculated in the deformation process [24], which supplies the forces and force directions at each material point. As a stress tensor is a represented by a 3x3 symmetric matrix, it has three real eigenvalues that correspond to the size of the principal stresses (with eigenvectors corresponding to stress directions), where positive eigenvalues relate to tensile forces and negative eigenvalues to compressive forces. The largest tensile eigenvalue is found for all tetrahedra in the material and these are compared against the stress threshold. If the threshold is exceeded, all tetrahedra within a set radius are divided either side of a fracture plane that is perpendicular to the eigenvalue's respective eigenvector.

IV. SIMULATION FRAMEWORK OVERVIEW

Building on our previous work [25] the framework presented here is constructed on top of the DirectX 10 API with programmable shaders in the HLSL [11] shading language. The framework includes three modules: deformation, fracture and procedural generation of the edges of destroyed materials. We use a tetrahedral based FEM incorporating the linear elasticity model and integrate the motion using the explicit Euler method [6]. Disregarding the effects of plasticity and the issues of warping under large rotational deformations, deformation and fracture in our system can selectively be performed on the CPU or the GPU. The descriptions of these two modules below focusses on the GPU implementation and describes the construction techniques used for the migration of processing from CPU to GPU.

A. Deformation Simulation

We employ a finite element method (FEM) [26] to model the deformation using tetrahedral elements with linear basis functions that produce a piecewise constant strain field over the elements. Objects are made up of connected cubes composed of five tetrahedra each: a central isosceles tetrahedron surrounded by four corner tetrahedra. Adjacent cubes in the object share tetrahedra nodes (corner points) and therefore use a setup similar to a mass-spring system. The deformation simulation is split into two stages: updating the finite elements and updating the nodes, ensuring that each finite element and each node are processed only once. After any external forces are applied to the nodes, the finite element update stage calculates the internal stresses and strains across a tetrahedron's faces and produces the internal forces to be applied to each node. The node update stage then applies the internal forces to update the node velocities and positions and integrates them over time to simulate the correct motion for the deformable material. The finite elements and nodes are represented as compactly as possible in order to minimise memory bandwidth usage. Where possible, variables are packed into float4 types to improve performance [27]. A single vertex buffer is created containing the list of finite elements and two vertex buffers with stream out capability are created containing a duplicate list of nodes that are used to switch between inputs and outputs in order to set up a feedback loop in the shader updates. A 128-bit floating point RGBA texture containing 32 bit elements is created to store the internal forces generated by the finite element method update, which are in turn used to update the accelerations of the nodes. These forces are stored in the RGB (Red/Green/Blue) elements and the remaining A (Alpha) element is used to store the inverse mass value for each node. This allows the anchoring of individual nodes if their corresponding value is set to zero. Using the spare alpha channel in this manner saves the overhead of storing the inverse mass value within each node structure. Each material has a set of configurable parameters that are passed to the shaders each frame via shader variables as they remain constant across all finite elements and nodes within a particular material. The 6x6 material stiffness matrix has its elements spread across nine float4 elements and each of these are passed to the shaders each frame via shader variables.

1) Deformation Finite Element Update Method: The finite element update method (Figure 2) takes the finite element vertex buffer and the current input node vertex buffer as inputs. A single point primitive is drawn for each finite element meaning that the entire method is invoked once per finite element. The finite element update stage processes each finite element tetrahedron as follows:

• Get the four node positions of the tetrahedron.
• Calculate the derivatives of the displacement field from the node positions.
• Calculate Green's strain tensor.
• Stress equals the material stiffness matrix times the strain.
• For each tetrahedron face, force equals stress times the face normal.
• Distribute tetrahedra face forces across tetrahedra nodes.

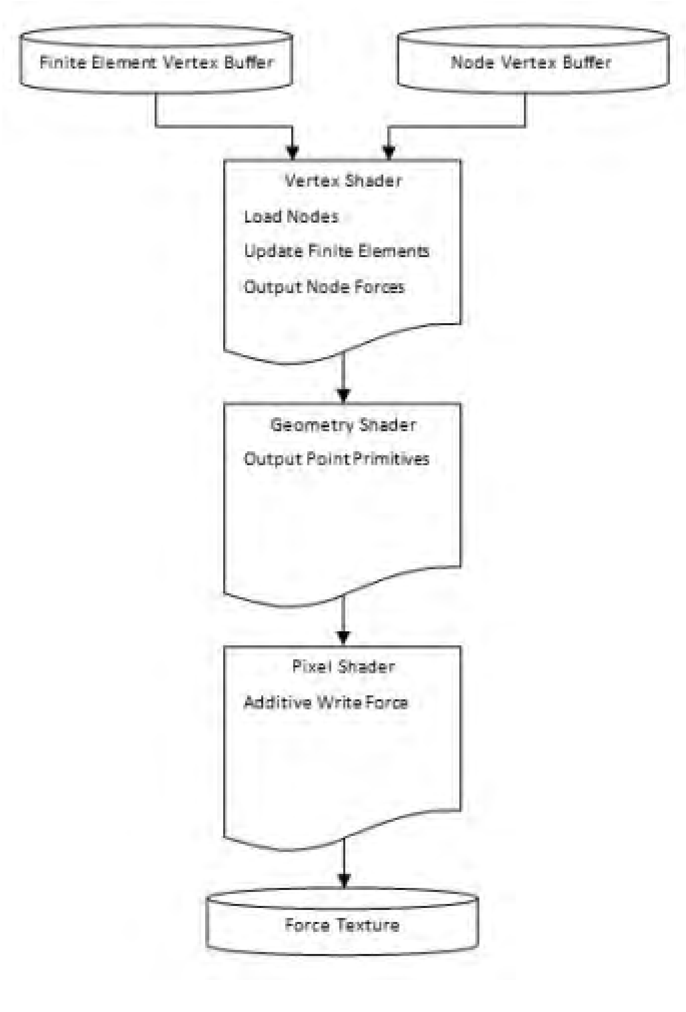

Fig. 2. Finite element update method (deformation).

The method is split into three shaders (Vertex Shader, Geometry Shader, and Pixel Shader), following the DirectX pipeline protocol with each shader feeding its outputs into the inputs of the

next. The vertex shader uses the HLSL Load function to get the positions of the four nodes for the finite element's tetrahedron. The finite element is then updated, producing four node forces (one for each of the nodes of the current tetrahedron). These four node forces are output to the geometry shader, which outputs a point primitive per force that are mapped to individual texels to be drawn into a floating point force texture. The force texture has one element per node and therefore the output texture coordinates are calculated using the node index relating to the force being output. The pixel shader draws the force texels into the force texture using an additive blend mode to accumulate them as it is possible that multiple forces per node could be generated by adjacent tetrahedra. The alpha write is disabled via the blend mode in order to preserve the inverse mass values that are stored in the force texture alpha channel. The output is then routed to a floating point texture instead of a vertex buffer in order to take advantage of the hardware accelerated additive blend mode available on the GPU.
2) The Node Update Method: The node update method (Figure 3) takes the current input node vertex buffer and the force texture as inputs. The vertex shader loads the forces from the force texture via a HLSL texture load (which also contains the inverse mass in the alpha channel). The texture coordinates used for the load are calculated using the current node index which is supplied automatically into the vertex shader via the SV VertexID semantic [28]. The nodes' positions and velocities are then updated and the stream out mechanism is used to output the updated node into the output node vertex buffer.

The node update stage processes each node as follows:
• Acceleration equals the force divided by the mass.
• Add gravity to the acceleration.
• Update the velocity by the time step times the acceleration.
• Update the position by the time step times the velocity.
• Add damping to the velocity.

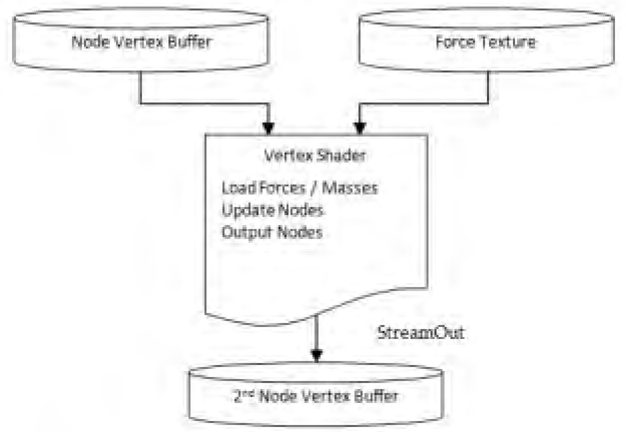

Fig. 3. Node update method.

The input and output node vertex buffers are then swapped for the following frame providing a feedback loop. The two vertex buffers are required to be setup in this manner as the DirectX API does not allow a single vertex buffer to be used as both input and output at the same time.

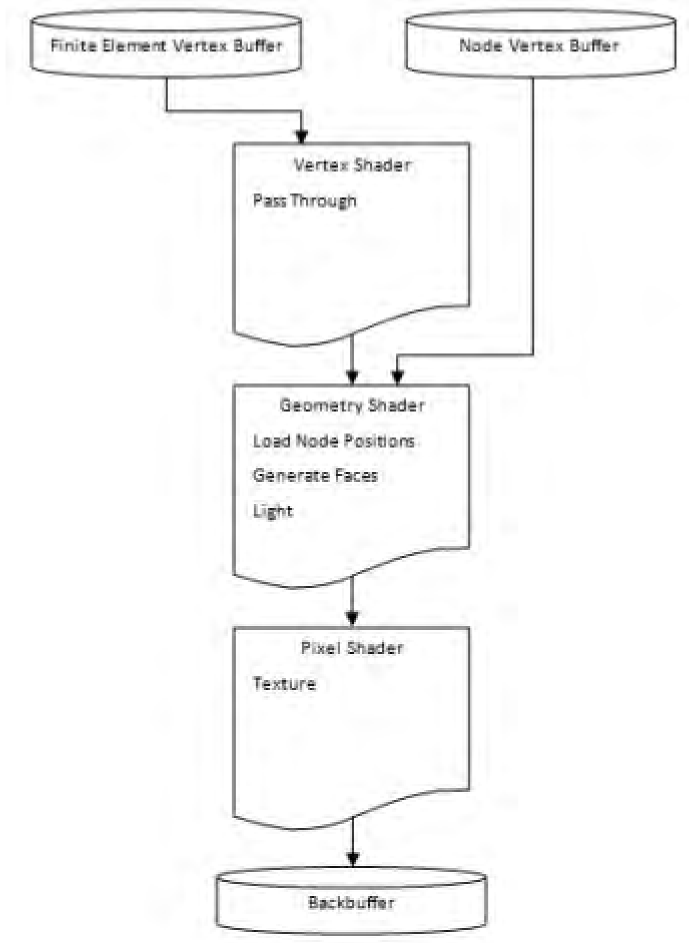

Fig. 4. Draw method.

3) The Draw Method: The draw method (Figure 4) takes the finite element vertex buffer and the current input node vertex buffer as inputs. A point primitive for each finite element is passed untouched through the vertex shader into the geometry shader. The geometry shader loads the node positions from the node vertex buffer and generates four triangle faces per finite element tetrahedron. The face normal and lighting based on a global light direction are calculated and the elements are passed to the pixel shader for texturing and output to the backbuffer of the framebuffer.

B. Fracture Simulation

We use a process similar to that described by Muller et al. [24] to compute the maximum eigenvalues for the finite elements in the simulation, which are then compared against the maximum stress threshold set for the current material to determine where cracks should occur. Each tetrahedron-based cube is broken away from the rest of the object structure in its entirety rather than splitting and retesselating at run time in order to maximise performance and to allow a GPU implementation.

Unfortunately, in a GPU stream processing architecture, duplicating shared nodes where the tetrahedron-based cubes meet each other is impossible to implement. This is because the finite elements and nodes are processed individually using streams of data as their input, rather than having the access to random elements that would be available within a CPU implementation.

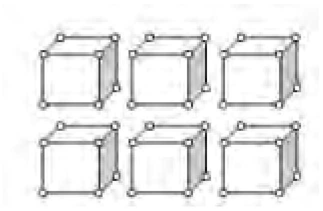

Fig. 5. Our new finite element connectivity.

Our solution to this problem changes the underlying structure for the connectivity of the nodes fed into the deformation system. Instead of the finite element based cubes being connected to the nodes in a sharing fashion similar to a mass spring system, each tetrahedron-based cube has its own unique set of eight corner nodes (Figure 5). A separate list of connections is stored for the eight corner nodes detailing the other corner nodes that the cube is currently connected to, which maintains a consistent state when updating the deformation simulation. After the forces are calculated in the deformation pass, a further pass accumulates each force based on its current connections. With the simulation operating in this manner, it is possible to flag a finite element as being disconnected while processing the individual finite element in the GPU vertex shader without the need to duplicate any nodes. When a finite elements corner node is flagged as disconnected, its forces are no longer accumulated with its neighbours and therefore it begins to separate itself from the adjacent elements. Building upon the resources allocated for the deformation module, a 128-bit floating point RGBA texture is created containing 32 bit elements to store the eigenvectors (in the RGB channels) and eigenvalues (in the Alpha channel). A second finite element vertex buffer is also allocated in order to setup a feedback loop for the finite elements in the fracture update method. The material structure is also modified, adding a maximum stress value for the material being simulated.

1) Fracture Finite Element Update Method: We use an improved version (Figure 6) of our finite element update method (Figure 2) to calculate the eigenvalues within the vertex shader, which are then passed through to the geometry shader where a point primitive is output for each finite element. The values are written out to the eigen-texture in the pixel shader using the MRT technology. The output texture coordinates are calculated using the automatically supplied SV VertexID semantic and therefore output to different locations than the output force values within the same geometry shader.

2) The Fracture Update Method: The fracture update method (Figure 7) takes the current input finite element vertex buffer and the eigen texture as inputs. The vertex shader loads the eigenvalues and marks the finite element as detached if the eigenvalue exceeds the maximum material stress value. The stream out mechanism is used to output the updated finite elements into the output finite element vertex buffer and the input and output finite element vertex buffers are then swapped for the following frame providing a feedback loop.

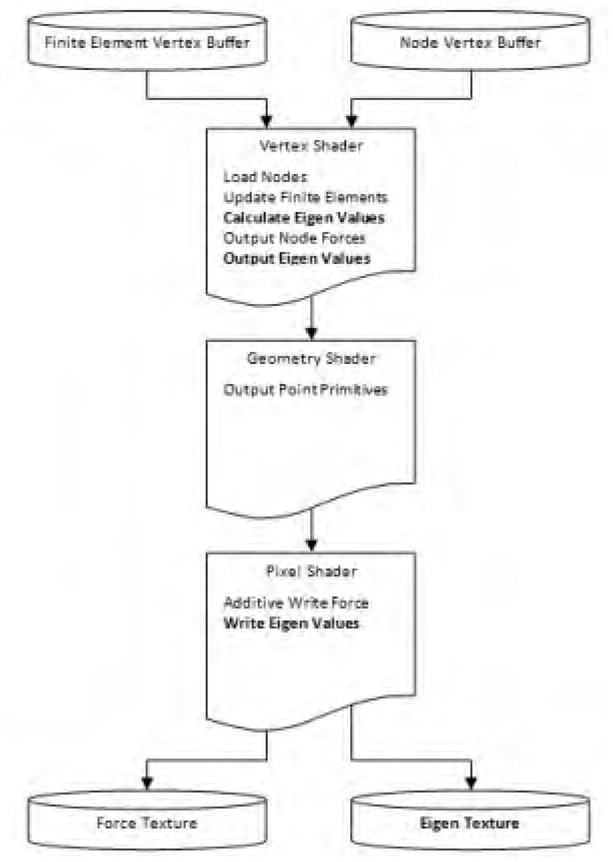

Fig. 6. Finite element update method (fracture).

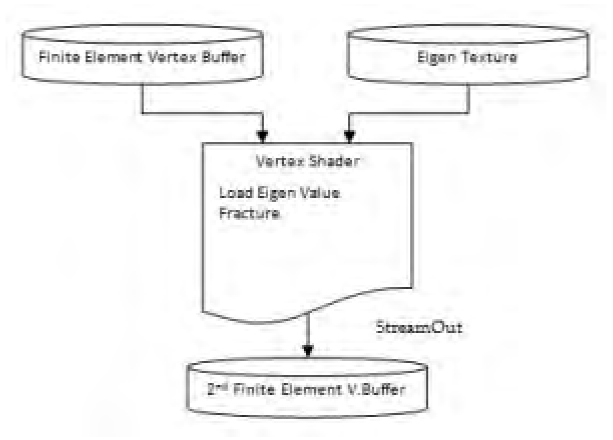

Fig. 7. Fracture update method.

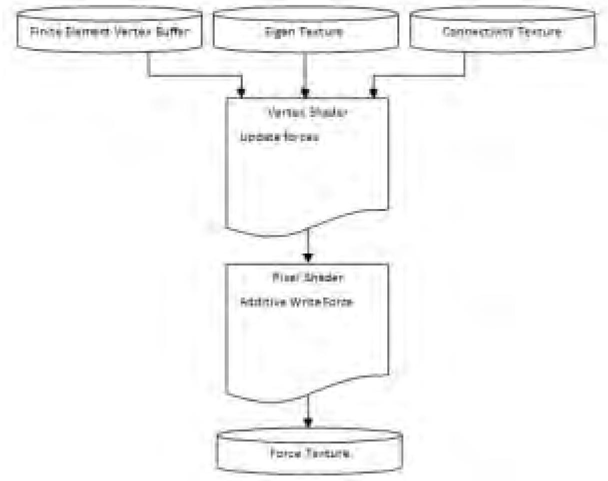

Fig. 8. Connectivity update method.

3) The Connectivity Update Method: The connectivity up-date method (Figure 8) takes the current input finite element vertex buffer as input along with a second eigen-texture to create a feedback loop for the eigen-texture. Also a connectivity texture is used as input that contains the connectivity information for each node. The vertex shader loads the finite element and for each node checks the connectivity information. If the node is connected, the forces are accumulated for that node and output to the pixel shader to be written out to the eigen-texture for updating the nodes positions and velocities in the deformation simulation.

C. Procedural Edges Simulation

Our method for providing interesting detail to the edges of fractured materials is inspired by the work of Scheepers and Whittock [29], where sections of geometry are precomputed to contain broken edge details. These are placed together so as to look undamaged in their initial state. A separate display mesh is created with each section relating to one tetrahedral cube within the simulation mesh. The vertex positions of the display mesh are linked to the finite elements in the simulation mesh using barycentric coordinates, allowing the display mesh movement to mirror the simulation mesh movement in real time.

Our solution for creating the look and feel of a destroyed material when the sections are fractured adjusts width and height of sections by a pseudo-random factor linked to a material parameter to provide user control. Additionally, a number of vertices are randomly placed along each section edge and attached to the edge with polygons. This allows the simulation of a wide range of material types by adjusting the material parameters. Display meshes are generated in a stochastic manner, so that each mesh is constructed differently from the others while still obeying the material parameters of the selected material type. Effectively this means that each time a material is broken apart at run-time, it looks and acts differently even though the display mesh was precomputed (Figure 1).

The initial display mesh is created with section sizes based on the material's average display section width and average display section height parameters divided into the requested width and height of the entire object. These parameters are referred to as 'averages' because they are first divided into the object's dimensions and then adjusted to be composed of evenly spaced sections. For each section, all edges apart from those on the perimeter of the object are adjusted based on the random display section width and random display section height material parameters. This has the effect of breaking up the uniform layout of the object's geometry and also creating a different layout each time the object is initialised. For each of the horizontal and vertical edges of each section excluding those around the walls perimeter, a number of vertices are inserted based on the horizontal edge frequency and vertical edge frequency material parameters. These are used to create polygon edges that stitch into the existing sections with their size based on the horizontal edge amplitude and vertical edge amplitude material parameters.

V. EVALUATION

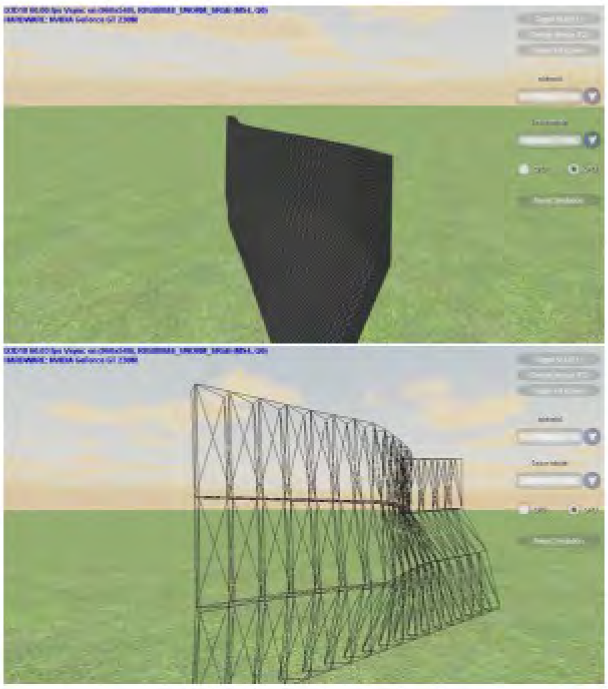

Fig. 9. Deformation on rubber material.

In our test implementation we simulated the deformation and fracture of wall-like objects on two platforms: an Intel R Core TM 2 Duo P7450 at 2.13GHz with 4 GB RAM and a GeForce GT 230M and an Intel R TM i7 920 at 2.66GHz with 8 GB RAM and an ATI Radeon HD 4870 X2. Frame rates for the GPU implementations reached a high of 56 FPS for the lower specification configuration and 110 FPS on the higher specification machine. In both configurations, the GPU implementations outperformed the CPU implementations by a factor of about 2:1.

Using different variations of material parameters along with a suitable display texture, our system allows the simulation of different materials for the wall object. For example, using square-shaped sections with few edge vertices provides the impression of a glass window when broken apart or using long thin sections without vertical edge vertices but with many horizontal edge vertices at a high amplitude gives the effect of planks of wood breaking off with sharp splinter-type edges. Figures 9 and 10 show deformation and fracture results on rubber and wood materials resulting from our system.

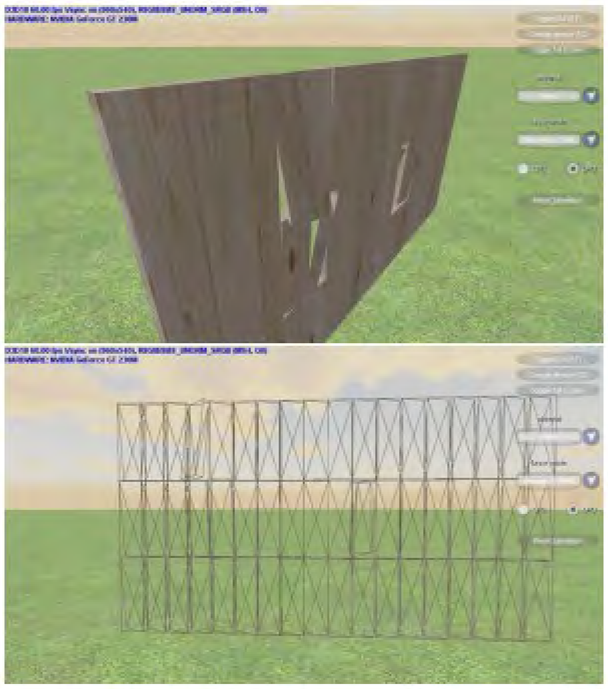

Fig. 10. Fracture on wood material.

VI. CONCLUSIONS AND FUTURE WORK

We have presented a system that allows objects to be deformed and fractured in a visually plausible manner, using interchangeable CPU and GPU techniques. The shader-based GPU implementations required the development of several novel approaches for data representation due to limits posed by the stream processing architecture of the GPU. The modular combination of CPU and GPU implementations allow efficient load balancing by spreading the processing requirements across all available processors, demonstrating the suitability of the system for use in a video game or other real-time environment. For example, in a video game requiring hundreds of fences to be simulated within a scene, simulation instances could be run both on the CPU and the GPU. To further evaluate the effectiveness of our system, future work will include quantitative comparison with the GPGPU approach. It should be noted that using the explicit Euler integration caused some instability and in order to keep the simulation from becoming unstable in both deformation and fracture simulations, material parameters had to be tuned very finely and only work within strict limits. The biggest improvement that could be made to our system would therefore be to implement a more advanced numerical integration method such as the implicit Euler method, which would not only add stability to the simulation but should allow a larger number of materials to be simulated, or possibly a hybrid method as presented by Fierz et al. [30].

The procedural edges methods presented here provide an initial step towards an automated system for producing edge geometry for fracture simulation. They could be improved by introducing more complex parameter controls such as providing a step to the edges for materials such as brick and matching these stepped edges with an appropriate texture to provide a convincing brick edge. For a full deformation and fracture simulation, two methods of collisions would need to be integrated into the simulation: object to object collisions for external objects colliding with the deformation object, and self-collisions for broken sections colliding with other sections of the object. A key contribution of this paper is the implementation of the simulation within shader programs utilising the advanced features of recent GPU's, rather than employing General Purpose GPU approach. Furthermore, the modular approach described here provides the potential to load-balance between CPU and GPU, an important issue as more and more tasks are migrated onto the GPU and especially important for applications that may bottleneck the GPU or where load changes may occur at runtime.

REFERENCES

[1] E. G. Parker and J. F. O'Brien, "Real-time deformation and fracture in a game environment," in SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2009, pp. 165–175.

[2] J. Van Gestel, "Procedural destruction of objects for computer games," Master's thesis, TU Delft, 2011.

[3] A. Garg and K. Maxwell, "Seamless fracture in a production pipeline," in ACM SIGGRAPH 2010 Talks, 2010, pp. 23:1–23:1.

[4] B. Cole, "Kali: high quality fem destruction in zack snyder's sucker punch," in ACM SIGGRAPH 2011 Talks, 2011, pp. 40:1–40:1.
[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," ACM Trans. Graph., vol. 23, no. 3, pp. 777–786, 2004.

[6] M. Muller, J. Stam, D. James, and N. Thurey, "Real time physics: class notes," in SIGGRAPH '08: ACM SIGGRAPH 2008 classes, 2008, pp. 1–90.

[7] J. Clark, "The Geometry Engine: A VLSI geometry system for graphics," in SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques, 1982, pp. 127–133.

[8] K. Akeley, "RealityEngine Graphics", in SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 1993, pp. 109–116.

[9] T. Akenine-Moller, E. Haines, and N. Hoffman, Real-Time Rendering, 3rd ed. A.K. Peters, 2008.

[10] W. Mark, R. Glanvillle, K. Akeley, and M. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," in SIGGRAPH '03: Proceedings of the 30th annual conference on Computer graphics and interactive techniques, 2003, pp. 896–907.

[11] Microsoft MSDN, "HLSL (Windows)," Available on-line at http://msdn.microsoft.com/en-us/library/bb509561(v=vs.85).aspx, 2010.

[12] R. Rost, OpenGL Shading Language, 2nd ed. Addison-Wesley, 2006.

[13] M. Joselli, E. Clua, A. Montenegro, A. Conci, and P. Pagliosa, "A new physics engine with automatic process distribution between cpu-gpu," in Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, ser. Sandbox '08, 2008, pp. 149–156.

[14] F. Liu, T. Harada, Y. Lee, and Y. J. Kim, "Real-time collision culling of a million bodies on graphics processing units," ACM Trans. Graph., vol. 29, pp. 154:1–154:8, December 2010.

[15] M. Tang, D. Manocha, J. Lin, and R. Tong, "Collision-streams: fast gpu-based collision detection for deformable models," in Symposium on Interactive 3D Graphics and Games, 2011, pp. 63–70.

[16] G. Aldrich, D. Pinskiy, and B. Hamann, “Collision-driven volumetric deformation on the GPU,” in Eurographics 2011 - Short Papers. Eurographics Association, 2011, pp. 9–12.

[17] Pixelux Entertainment, “Digital Molecular Matter,” Available on-line at http://www.pixelux.com, 2010.

[18] LucasArts, “Star Wars: The Force Unleashed,” Available on-line at http://www.lucasarts.com/games/theforceunleashed, 2008.

[19] Nvidia, “APEX destruction,” Available on-line at http://developer.nvidia.com/apex-destruction, 2012.

[20] M. Muller and M. Gross, “Interactive virtual materials,” in Proceedings of Graphics Interface 2004, 2004, pp. 239–246.

[21] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, “Elastically de-formable models,” in SIGGRAPH ’87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987, pp. 205–214.

[22] J. F. O’Brien and J. K. Hodgins, “Graphical modeling and animation of brittle fracture,” in SIGGRAPH ’99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, 1999, pp. 137–146.

[23] J. F. O’Brien, A. W. Bargteil, and J. K. Hodgins, “Graphical modeling and animation of ductile fracture,” ACM Trans. Graph., vol. 21, no. 3, pp. 291–294, 2002.

[24] M. Muller, L. McMillan, J. Dorsey, and R. Jagnow, “Real-time simulation of deformation and fracture of stiff materials,” in Eurographics workshop on Computer animation and simulation, 2001, pp. 113–124.

[25] D. J. Morris and E. F. Anderson, “GPU destruction: Real-time procedural demolition of virtual environments,” in Eurographics 2010 – Posters, 2010.

[26] B. Donald, Practical Stress Analysis with Finite Elements. Glasnevin Publishing, 2007.

[27] Nvidia, “GPU programming guide GeForce 8 and 9 series,” Available on-line at http://developer.nvidia.com/object/gpu programming guide.html, 2008.

[28] Microsoft MSDN, “Semantics (DirectX HLSL),” Available on-line at http://msdn.microsoft.com/en-us/library/bb509647(VS.85).aspx, 2010.

[29] F. Scheepers and A. Whittock, “The wrecked road in Cars - or how to damage perfectly good geometry,” in SIGGRAPH ’06: ACM SIGGRAPH 2006 Sketches, 2006, p. 97.

[30] B. Fierz, J. Spillmann, and M. Harders, “Element-wise mixed implicit-explicit integration for stable dynamic simulation of deformable objects,” in Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium in Computer Animation, 2011, pp. 257–266.